Appendix C

Following is a description of a parallel port interface that gives full access to the all the parallel port pins and implements a parallel port data transfer functionality that can be used in conjuction with the ESL download utility

```
// ************************************************************
// Parallel port controller
// ************************************************************

// Instantiates a component that controls the parallel port.
// This is to be run in parallel in the main loop. The interfaces
// provide the user with abstracts to use deal efficiently with the
// component.

// ****************************************
// Interfaces
//
// API to Parallel Port - for direct access to the pins
//
// PpWriteData((unsigned 8)byte) -- write byte to data pins
// PpReadData((unsigned 8)byte) -- read byte from data pins
// PpReadControl((unsigned 4)control_port) -- read the control port
// PpReadStatus((unsigned 6)status_port) -- read the status port
// PpSetStatus((unsigned 6) status_port) -- write to the status port
//
//
// API for the ESL parallel data transfer utility
//
```

```
// OpenPP(error) -- open the parallel port for data transfer
// ClosePP(error) -- close the port
// SetSendMode(error) -- set the port to send mode
// SetRecvMode(error) -- set the port to receive mode
// SendPP(byte, error) -- send a byte over the port
// ReadPP(byte, error) -- read a byte from the port
//
// error returns the result of the command:
// 0 - no error
// 1 - buffer error
// 2 - timeout error
//
// Note: SendPP and ReadPP will block the thread until  a byte is transmitted or the
timeout
// value is reached. If you need to do some processing while waiting for a
communication
// use a 'prialt' statement to read from the global pp_recv_chan channel or write to the
// pp_send_chan channel.


////////////////////////////////////////////////////////////////////////////
//  The Nitty Gritty
////////////////////////////////////////////////////////////////////////////


// The necessary channels
chan unsigned 8 pp_send_chan, pp_recv_chan;
chan unsigned 2 pp_command, pp_error;
```

```
chan pp_data_send_channel, pp_data_read_channel, pp_control_port_read;
chan pp_status_port_read, pp_status_port_write;


#define OPEN_CHANNEL   0
#define CLOSE_CHANNEL 1
#define SEND_MODE              2
#defineRECV_MODE               3

#define PP_NO_ERROR            0
#define PP_HOST_BUFFER_NOT_FINISHED     1
#define PP_OPEN_TIMEOUT 2

// Currently the functions don't act on any errors, but this can easily be added if
required.
// return of error code could also be used to generate a time-out condition.


macro proc OpenPP(error)
{
        pp_command ! OPEN_CHANNEL;
        pp_error ? error;
}


macro proc ClosePP(error)
{
        pp_command ! CLOSE_CHANNEL;
        pp_error ? error;
```

```
}

macro proc SetSendMode(error)
{
        pp_command ! SEND_MODE;
        pp_error ? error;
}

macro proc SetRecvMode(error)
{
        pp_command ! RECV_MODE;
        pp_error ? error;
}


macro proc WritePP(byte, error)
{
        pp_send_chan ! byte;
}


macro proc ReadPP(byte, error)
{
        pp_recv_chan ? byte;
}

// ************************************************************
```

```
// Parallel port controller
// ************************************************************

// Host Channel Control (HCC)   nAutoFeed
// FPGA Channel Control (FCC)   DONE
// Host Data Control (HDC)      nSelect_in
// FPGA Data Control (FDC)      nACK
// FPGA ready to communicate (FRTC)  PE


// HCC indicates that host is sending -  end of the buffer
// FCC controls direction of commmunication
// FRTC indicates that FPGA is ready
// when FPGA sets FCC low, rising edge on FDC when data applied
// lower when host responds with HDC high
// when FCC high FPGA is in receive mode and host applies data
// on rising edge on HDC. FPGA responds with FDC high and host
// then lowers HDC. Host will keep data byte on pins till FDC is
// lowered again by the FPGA

// chan unsigned 8 pp_data_chan;
// chan unsigned 4 pp_control_chan;
// chan unsigned 5 pp_status_chan;


////////////////////////////////////////////
// Macro to implement ESLs bi-directional host-fpga
// data transfer protocol
```

```
// Accesses the physical layer
//////////////////////////////////////////////

5

macro proc Test_PP()
{

        unsigned 4 control_port;
10      unsigned 6 status_port;

        unsigned 21 counter;

    //  PpSetControl(0b0000);
15      PpSetStatus(0b000000);

        do
        {
        counter++;
        }while(counter != 0);
20

        PpSetStatus(0b000001);

        do
25      {
        counter++;
        }while(counter != 0);

        PpSetStatus(0b000010);
```

```
do
{
counter++;
}while(counter != 0);


PpSetStatus(0b000100);


do
{
counter++;
}while(counter != 0);


PpSetStatus(0b001000);

do
{
counter++;
}while(counter != 0);


PpSetStatus(0b010000);

do
{
counter++;
}while(counter != 0);
```

```
PpSetStatus(0b000000);

        do
{
counter++;
}while(counter != 0);

PpSetStatus(0b011111);

while(1)
{
        PpReadControl(debug_control);
}
}

macro proc pp_coms(pp_send_chan, pp_recv_chan, pp_command, pp_error)
{

        // bit masks for accessing control and status ports

//control_port = nSelect_in.in @ init.in @ nAutofeed.in @ nStrobe.in;
#define HCC control_port[1] //0b0010   //nAutofeed pin on control port
```

```
#define HDC control_port[2] //0b0100   //nInit pin on control port

//status_port = ppdir @ busy @ nAck @ pe @ select @ nError;
#defineFRTC 0b000010              //pe pin on status port
#define FCC    0b000100           //select pin on status port
#define FDC    0b001000           //nAck pin on status

#definePP_SEND 0b100000
#define PP_READ 0b000000


        unsigned 4 control_port;
        unsigned 6 status_port;
        unsigned 1 pp_dir with {warn = 0};
        unsigned 2 command;
        unsigned 8 temp_data;

        PpSetStatus(PP_READ | FRTC); // initialise the port, read mode, FRTC high

        while(1)
        {
                prialt
                {
                                case pp_command ? command:

                                        // deal with any commands received
                                        switch (command)
                                        {
                                        case OPEN_CHANNEL:
```

```
// open channel and set to FPGA send
mode
PpSetStatus(PP_SEND | FCC ); // |FDC
keep FCC low, FRTC low to indicate ready
pp_dir = 1;

// wait for pulse on HCC in response to
open channel
PpReadControl(control_port);
while(HCC) // wait for nHCC to go low
{
        PpReadControl(control_port);
}
while(!HCC) // wait for nHCC to go high
{
        PpReadControl(control_port);
}
```

```
                pp_error ! PP_NO_ERROR;

                break;

5

            case CLOSE_CHANNEL:  // closes the channel
    regardless of state
                PpSetStatus(PP_READ  | FRTC); // sets
10  status port to all zeros, FRTC high
                pp_dir = 0;
                pp_error ! PP_NO_ERROR;
                break;

15

            case SEND_MODE:

                PpReadControl(control_port);

20

                // set FRTC  high - host send, start driving
    data pins, FCC low
                PpSetStatus(PP_SEND);
25              pp_dir = 1;
                pp_error ! PP_NO_ERROR;

                // BUFFERNOTFINISHED
                break;
```

```
                                case RECV_MODE:

                                        // set FRTC  high - host read - stop driving
    data pins, FCC high, FDC low

                                        PpSetStatus(PP_READ | FCC );
    //|FDC|FCC

                                        pp_dir = 0;
                                        pp_error ! PP_NO_ERROR ;

                                        break;


                                default:
                                        delay;
                                        break;
                                }

                                break;


                                // FPGA sending
                                case pp_send_chan ? temp_data:


                                        PpSetStatus(PP_SEND); // FCC low, FDC
    low - pin is inverted
```

```
PpReadControl(control_port);

while(!HCC) // wait for host to de-assert
```

HCC

```
{
        PpReadControl(control_port);
}

PpWriteData(temp_data);
PpSetStatus(PP_SEND | FDC);// FCC low,
```

FDC high

```
PpReadControl(control_port);


while(!HDC) // wait for host to assert HDC
{
        PpReadControl(control_port);
}


PpSetStatus(PP_SEND); // FCC low, FDC
```

low - pin is inverted

```
PpReadControl(control_port);

while(HDC) // wait for host to de-assert
```

HDC

```
			{
				PpReadControl(control_port);
			}

			break;

			// host sending
		default:

			PpReadControl(control_port);
			PpReadStatus(status_port);

			if (!status_port[5] & !HCC) // read one byte, if in read mode and HCC is low
			{

				while(!HDC)  // wait for host to apply data and raise HDC
				{
					PpReadControl(control_port);

				}
```

```
                                        PpSetStatus( PP_READ | FCC |
FDC); // FCC high FDC high

                                        PpReadData(temp_data);

                                        pp_recv_chan ! temp_data;

                                        PpReadControl(control_port);
                                        PpReadStatus(status_port);

                                        while(HDC) // wait for host to
remove HDC
                                        {
        PpReadControl(control_port);

                                        }

                                        PpSetStatus( PP_READ | FCC ); //
FCC high FDC low

                                    }
                                    else delay;

                                    break;

                    }

            } // while(1)
```

```
        delay; // avoid combinational cycles
    }


5


10  ////////////////////////////////////////////////////////
    // Parallel Port - Physical layer
    //
    // Allows access to all the data, control and status ports
    // through a series of channels which can be read from
15  // and written to.
    ////////////////////////////////////////////////////////

    // Macro abstractions for the various actions

20  macro proc PpWriteData(/*(unsigned 8)*/ byte)
    {
        pp_data_send_channel ! byte;

    }

25

    macro proc PpReadData(/*(unsigned 8)*/ byte)
    {
        pp_data_read_channel ? byte;
```

```
}

macro proc PpReadControl(/*(unsigned 4)*/ control_port)
{
        pp_control_port_read ? control_port;

}


macro proc PpReadStatus(/*(unsigned 6)*/ status_port)
{
        pp_status_port_read ? status_port;

}

macro proc PpSetStatus(/*(unsigned 6)*/ status_port)
{
        pp_status_port_write ! status_port;
}



// Actual Parallel Port control circuitry

macro proc parallel_port(pp_data_send_channel, pp_data_read_channel,
pp_control_port_read,
```

```
                                        pp_status_port_read,
pp_status_port_write)
{

    unsigned 8 pp_data;
    unsigned 6 status_register;

    interface  bus_ts_clock_in (unsigned 8) data_bus(pp_data, status_register[5])
with pp_data_pins;


    // Control Port (unsigned 4, made up as nSelect_in.in @ init.in @ nAutofeed.in
@ nStrobe.in)
    interface bus_clock_in (unsigned 4) control_port() with control_port_pins;


    //  Status Port, status_register = pp_direction @ busy @ nAck @ pe @ Select @
nError;
    interface bus_out() status_port_bus(status_register[4:0]) with status_port_pins;

    // Setting pp_direction to 1 will drive data onto the pins.

    while(1)
    {
        // Allows read of control, read / write of status and data ports
simulatneously
        par
        {
```

```
prialt
{
        case pp_control_port_read ! control_port.in:
                break;

        default:
                delay;
                break;
}


prialt
{
        case pp_status_port_write ? status_register:
                break;

        case pp_status_port_read ! status_register:
                break;

        default:
                delay;
                break;
}


prialt
{
        case pp_data_send_channel ? pp_data:
```

```
                        break;

                    case pp_data_read_channel ! data_bus.in:
                        break;

                    default:
                        delay;
                        break;
                }

            }

        }
        delay; // to avoid combinational cycles
    }


    //macro expr control_port = nSelect_in.in @ init.in @ nAutofeed.in @
nStrobe.in;

    /*interface  bus_clock_in (unsigned 1) nAutofeed() with nAutoFeed_pin;
    interface  bus_clock_in (unsigned 1) init() with init_pin;
    interface  bus_clock_in (unsigned 1) nSelect_in() with nSelect_in_pin;
    interface  bus_clock_in (unsigned 1) nStrobe() with nStrobe_pin;

    // defined in the same order as on a PC
```

```
        macro expr control_port = nSelect_in.in @ init.in @ nAutofeed.in @ nStrobe.in;
*/

        /*
        interface  bus_out () nAck_line( status_register[3] ) with nAck_pin;
        interface  bus_out () busy_line(status_register[4]) with busy_pin;
        interface  bus_out () pe_line(status_register[2]) with pe_pin;
        interface  bus_out () select_line(status_register[1]) with select_pin;
        interface  bus_out () nError_line(status_register[0]) with nError_pin;
        */

        // status_register[5] is high to send and low to receive
        // defined in the same order as on a PC
//      macro expr status_port = pp_direction @ busy @ nAck @ pe @ Select @
nError;
```